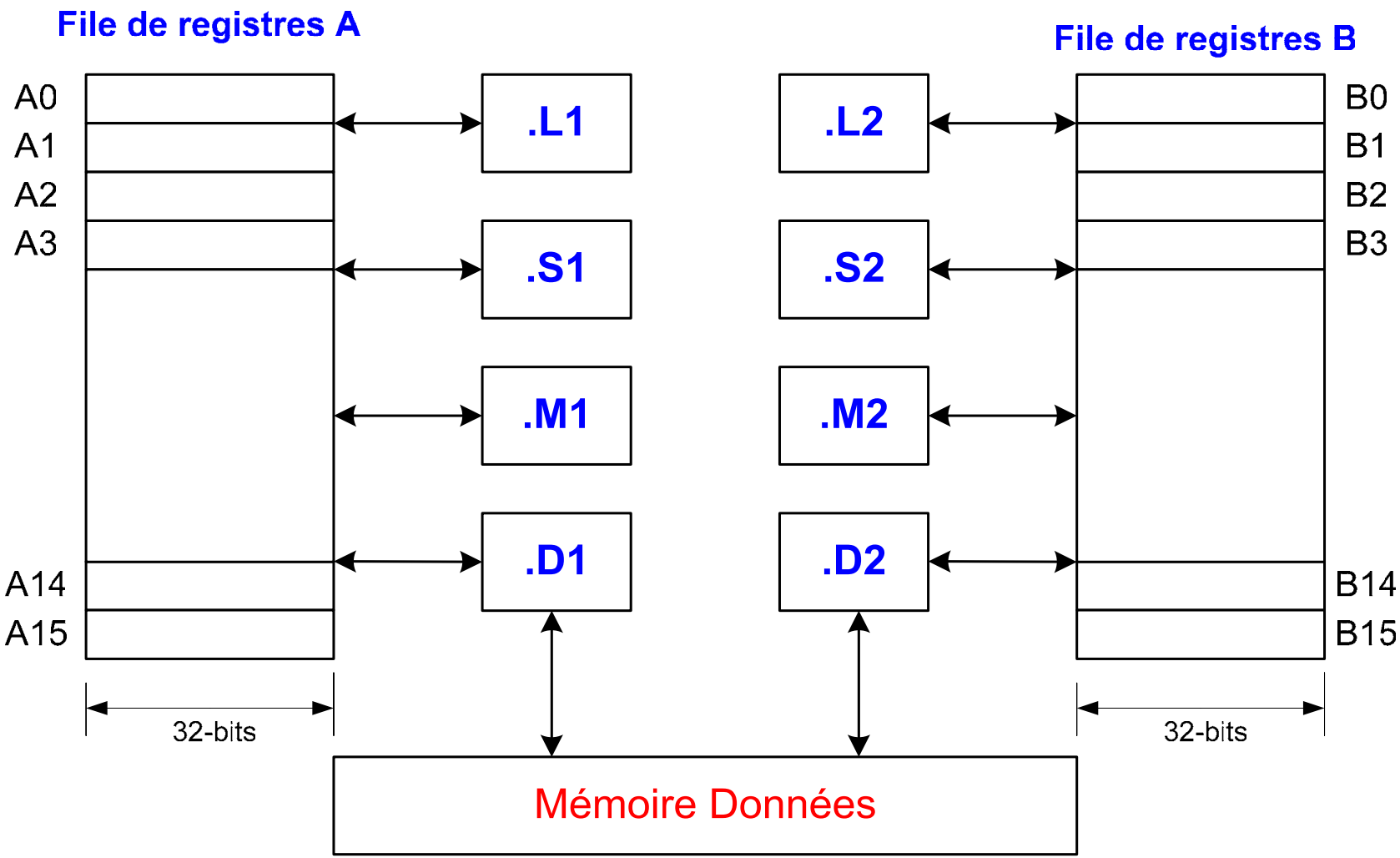
TMS320C6711

Un DSP VLIW

Caractéristiques

- Processeur 32 bits
- Adresses: 32 bits (Byte-addressable)
- Unité de calcul flottante: 600 MFLOPS
- VelociTI architecture (VLIW modifiée): parallélisme au niveau instruction.
- Comportement déterministe
- 2 multiplieurs, 6 ALUs
- Support matériel pour IEEE simple et double précision.

DSP core (TMS320C6711)



Unités fonctionnelles

- .M unité pour les multiplications
- .L unité pour les opérations logiques et arithmétiques
- .S unité pour les opérations de branchement, arithmétique et opérations sur bits
- .D unité pour chargement/rangement et arithmétique.

Produit scalaire

.L1

$$Y = \sum_{n=1}^{n=200} a_n \bullet x_n$$

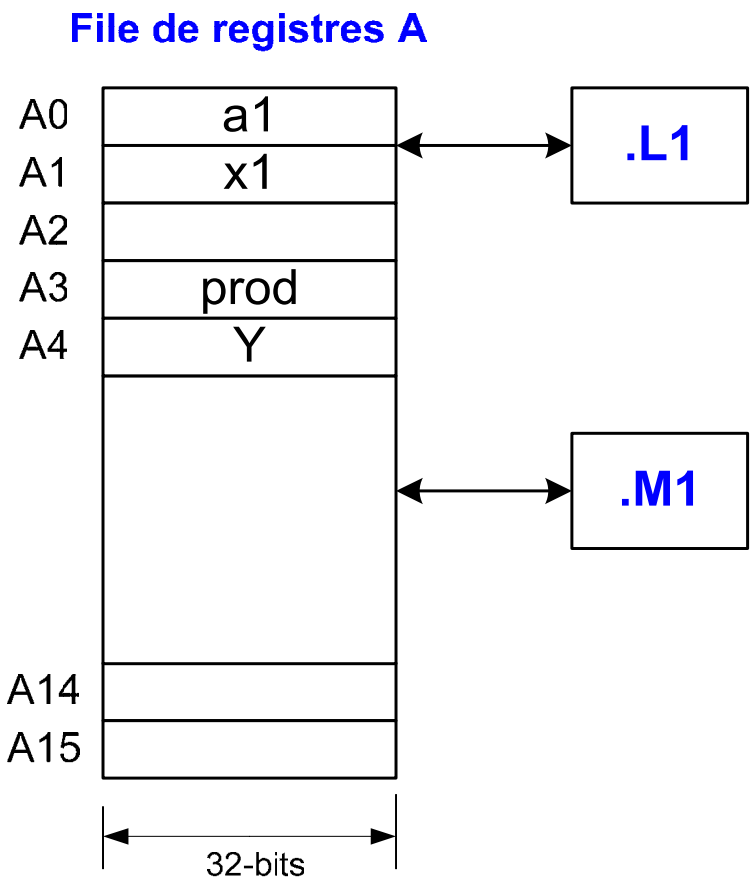
.M1

```
MPY     .M      a1,x1,prod
ADD     .L      Y,prod,Y
```

Nom de l'unité

Attention:
src1, src2, dest

Produit scalaire



$$Y = \sum_{n=1}^{n=200} a_n \bullet x_n$$

MPY	**.M**	**A0,A1,A3**
ADD	**.L**	**A4,A3,A4**

Nécessité de faire les opérations sur des registres

Chargement des registres

LD *Rn,Rm

Reg[Rm] ← MemD [Reg[Rn]]

Quelle taille?

LDB: one Byte (8 bits)

LDH: one Half word (16 bits)

LDW: one Word (32 bits)

LDDW: one Double word (64 bits)

Obligatoirement unité .D

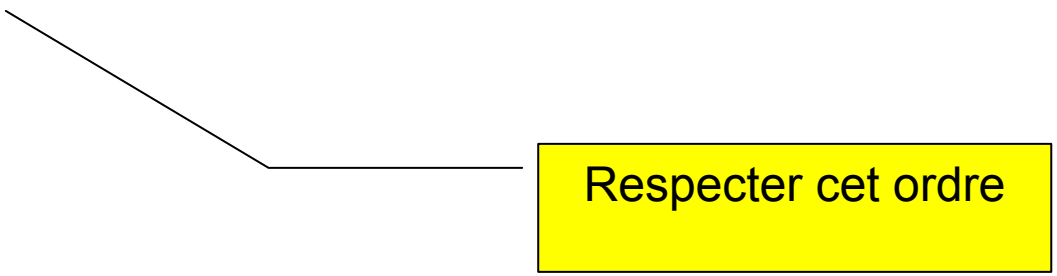
Chargement d'une adresse

Adresse: 32 bits

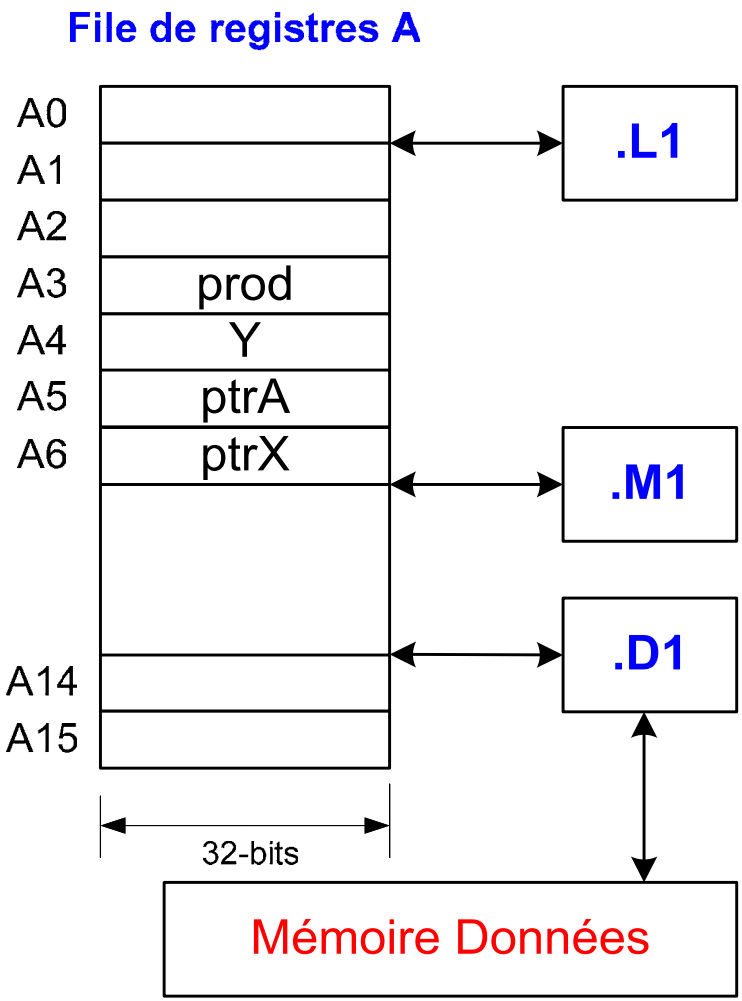
Instructions: 32 bits

D'où la nécessité de transmettre en deux fois :

```
MVKL    ptr,A5
MVKH    ptr,A5
```



Produit scalaire




```
MVKL    a_addr,A5
MVKH    a_addr,A5
MVKL    x_addr,A6
MVKH    x_addr,A6

LDH  .D  *A5,A0
LDH  .D  *A6,A1
MPY  .M  A0,A1,A3
ADD  .L  A4,A3,A4
```

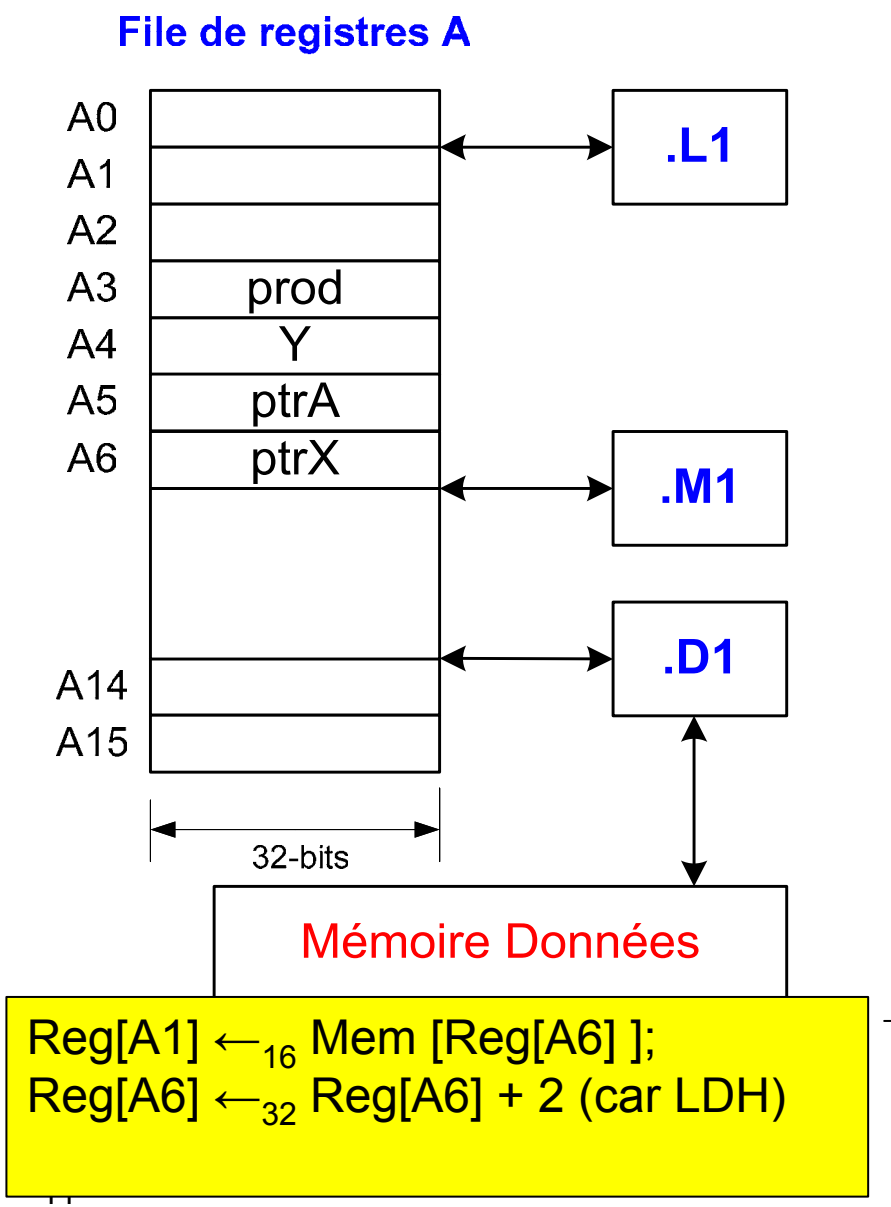
Manipulation des pointeurs

*R	Pointeur	inchangé
*+R [d]	+pre-décalage	inchangé
*-R [d]	-pre-décalage	inchangé
*++R [d]	Pre-incrémenté	modifié
*--R [d]	Pre-décrémenté	modifié
*R++ [d]	Post-incrémenté	modifié
*R– [d]	Post-décrémenté	modifié

d (déplacement) = 5-bit constante ou registre (utilisé comme index)

EA(*+R[d])=Reg[R]+k*d avec k = 1,2 ou 4 suivant les données

Produit scalaire




```
MVKL    a_addr,A5
MVKH    a_addr,A5
MVKL    x_addr,A6
MVKH    x_addr,A6

LDH .D  *A5++,A0
LDH .D  *A6++,A1
MPY .M  A0,A1,A3
ADD .L  A4,A3,A4
```

Boucle

1. Créer une étiquette (label)
2. Créer un compteur de boucle
3. Insérer une instruction pour décrémenter le compteur de boucle
4. Faire un branchement conditionné par la valeur du compteur de boucle

```
for ( k=N; k>0; k--) { … }
```

Instruction de branchement

Syntaxe: [condition] B label

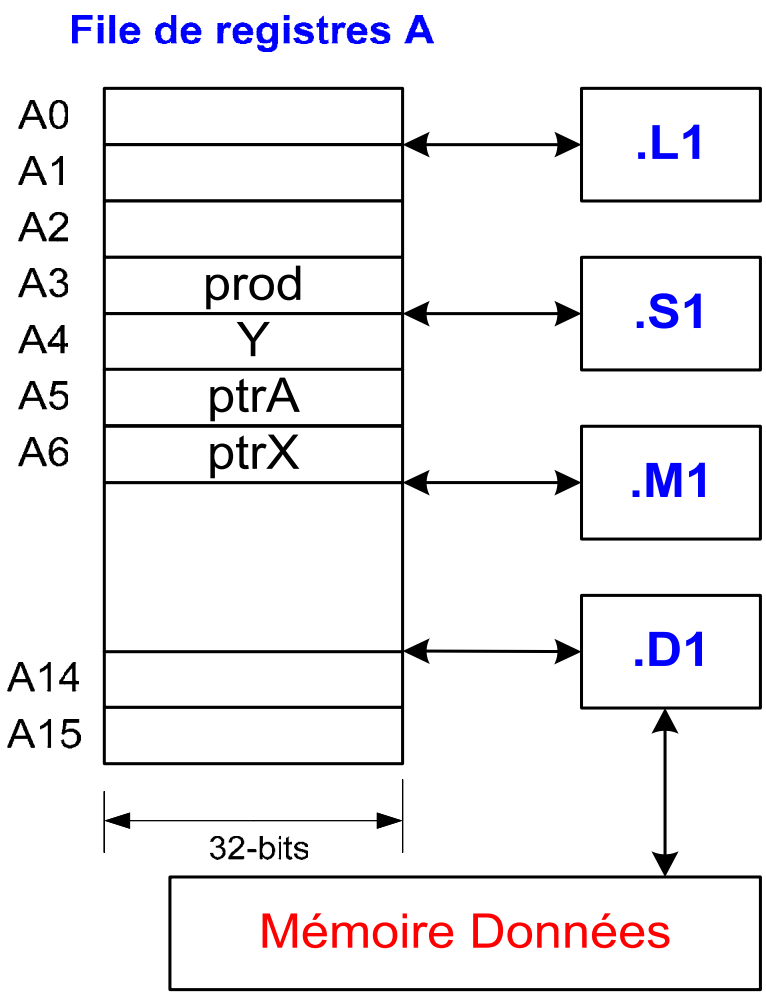
Adressage relatif:

- Adresse relative sur 21 bits
- Utiliser l'unité .S
- Syntaxe: B .S1 *label*

Adressage absolu:

- Adresse sur 32 bits
- Utiliser obligatoirement .S2
- Syntaxe: B .S2 *register*

Produit scalaire




```
                MVKL    .S      a_addr,A5
                MVKH    .S      a_addr,A5
                MVKL    .S      x_addr,A6
                MVKH    .S      x_addr,A6
                MVKL    .S      y_addr,A7
                MVKH    .S      y_addr,A7
                MVKL    .S      200,A2
                ZERO    .L      A4
bcl             LDH     .D      *A5++,A0
                LDH     .D      *A6++,A1
                NOP             4
                MPY     .M      A0,A1,A3
                NOP
                ADD     .L      A4,A3,A4
                SUB     .S      A2,1,A2
[A2]            B       .S      bcl
                NOP             5
                STH     .D      A4,*A7
```

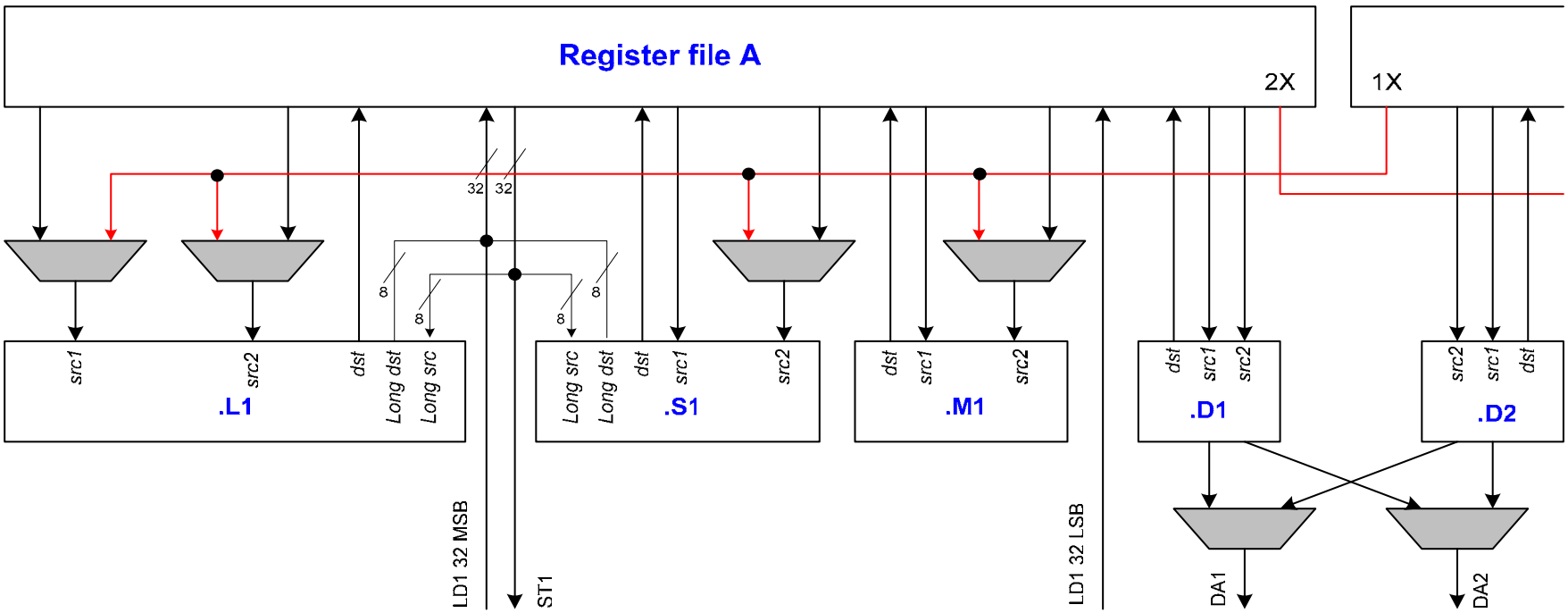
Delay Slots

- La plupart des instructions sont exécutées en 1 cycle, mais pas toutes!
- Après certaines instructions il faut rajouter des NOP:
- Les multiplications: 1 NOP
- Les instructions de chargement: 4 NOP
- Les instructions de branchement: 5 NOP
- Des instructions flottantes double précision…

Amélioration des performances

- On peut augmenter la fréquence d'horloge: réduit le temps de cycle et donc plus d'opérations par unité de temps
- Augmenter le parallélisme d'exécution en rajoutant des unités. Jusqu'à 8 instructions en parallèle.
- Pipeline
- Optimisation de code

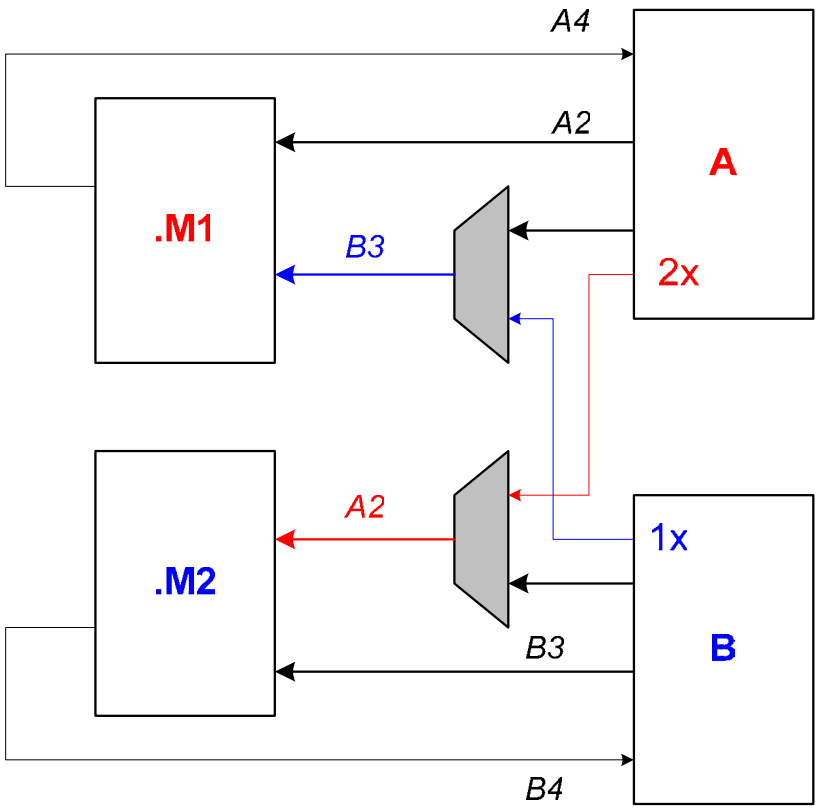
Data Cross Paths



Chemins croisés

- Un groupe d'unité peut accéder à un opérande dans la file de registre de l'autre groupe.
- Il y a des limitations.
- Cross Path
  - Data Cross Path (Registerfile cross path)
  - Address Cross Path
- Execute Packet (EP) = groupe d'instructions qui peuvent être exécutées en parallèle pendant le même cycle.
- Fetch Packet (FP) = 256 (8 words)

Data Cross-Path

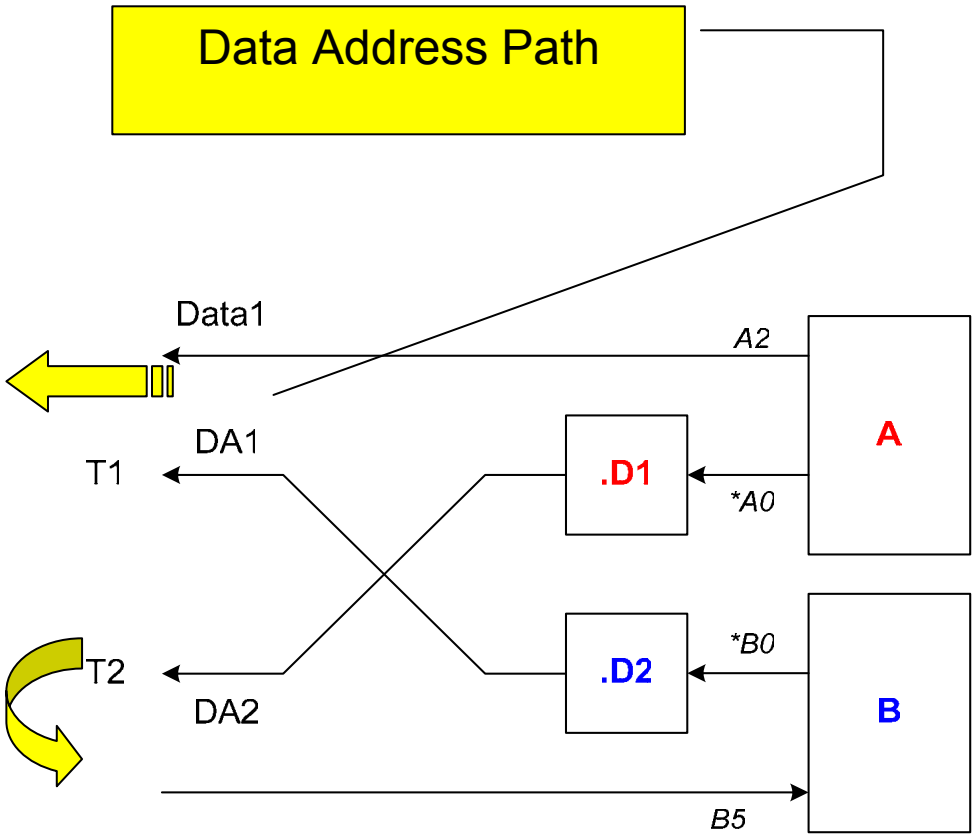


Data Cross

```
MPY  .M1x  A2,B3,A4
|| MPY .M2x  A2,B3,B4
```

Exécution parallèle

Address Cross-Path




```
LDW.D1T2    *A0,B5
|| STW.D2T1  A2,*B0
```

Cross Path Limitations

- Data Cross Path
  - Le registre de destination doit être du même côté que l'unité qui exécute
  - Pour les registres source: au plus un croisement par EP par côté.
  - Notation: x
- Address Cross path
  - Le pointeur doit être du même côté que l'unité
  - Une donnée peut être transférée depuis/vers chacun des côtés
  - Si accès parallèle: ou bien deux croisements ou aucun
- Pas de croisement avec les conditionnelles.

Produit scalaire optimisé (1)

```
        ZERO.L1     A4          ; Y ← 0
   ||   MVKL.S1     200,B0      ; cpt boucle k
Bcl     LDH.D1     *A8++,A2     ; A2 ← a(k)
   ||   LDH.D2     *B9++,B3     ; B3 ← x(k)
   [B0] SUB.L2     B0,1,B0      ; k - -
   [B0] B.S1       Bcl
        NOP        2
        MPY.M1x    A2,B3,A3     ; prod ← a(k)*x(k)
        NOP
        ADD.L1     A3,A4,A4     ; Y ← Y + prod

Done    STH.D1     A4,*A7       ; range le résultat
```

4 delay slots

Parallel execution

Delayed Branch of 5

Produit scalaire optimisé (2)

- Lorsque les données sont des halfwords (16 bits) on peut réaliser 2 chargements en une seule instruction puisque les registres sont des words (32 bits).
- Pour une multiplication on utilise
  - MPY (multiply 16 LSBs)
  - MPYH (multiply 16 MSBs)

Produit scalaire optimisé (2)

```
        ZERO.L1     A4          ; Y1 ← 0
   ||   ZERO.L2     B2          ; Y2 ← 0
   ||   MVKL.S1     100,B0      ; cpt boucle k ← 200/2

Bcl     LDW.D1     *A8++,A2     ; A2 ← a(2k)##a(2k+1)
   ||   LDW.D2     *B9++,B3     ; B3 ← x(2k)##x(2k+1)
   [B0] SUB.L2     B0,1,B0      ; k - -
   [B0] B.S1       Bcl
        NOP        2
        MPY.M1x    A2,B3,A2     ; prod1 ← a(2k+1)*x(2k+1)
   ||   MPYH.M2x   A2,B3,B3     ; prod2 ← a(2k)*x(2k)
        NOP
        ADD.L1     A4,A2,A4     ; Y1 ← Y1 + prod1
   ||   ADD.L2     B2,B3,B2     ; Y2 ← Y2 + prod2

Done    ADD.L1x    A4,B2,A5     ; Y1 + Y2
        STH.D1     A5,*A7       ; range le résultat
```

Deux fois moins de boucles

Charger 2 halfwords

Multiplications sur 16 bits

Performances

Comparaison des durées de boucle pour les 3 implémentations du produit scalaire

1. Non optimisé: 16 cycles par boucles, 200 itérations : 3200 cycles
2. Optimisé ||+ remplacement de NOP : 8 cycles par boucles, 200 itérations : 1600 cycles
3. Optimisé ||+ NOP + words : 8 cycles par boucles, 100 itérations : 800 cycles.

Peut-on faire mieux? OUI, pipeline logiciel

Pipeline des C6x

1. Etage « program fetch »
   1. PG: program address generate
   2. PS: program address send
   3. PW: program address ready wait
   4. PR: program fetch packet receive
2. Etage « decode »
   1. DP: dispatch instructions FP to functional units
   2. DC: instruction decode
3. Etage « execute »
   1. Jusqu'à 6 phases en virgule fixe
   2. Jusqu'à 10 phases en flottant double précision

Retard / latence

Delay slots: retards qu'il faut introduire pour que l'opération soit terminée

Latence: nombre de cycles pendant lequel une instruction est liée à une unité fonctionnelle

Exemple: MPYDP latence=4; durée=10 (dont 9 delays).

Durées:

1 pour la plupart des instructions

2 pour multiplication virgule fixe

5 pour un load

6 pour un branchement

...

Pipeline logiciel

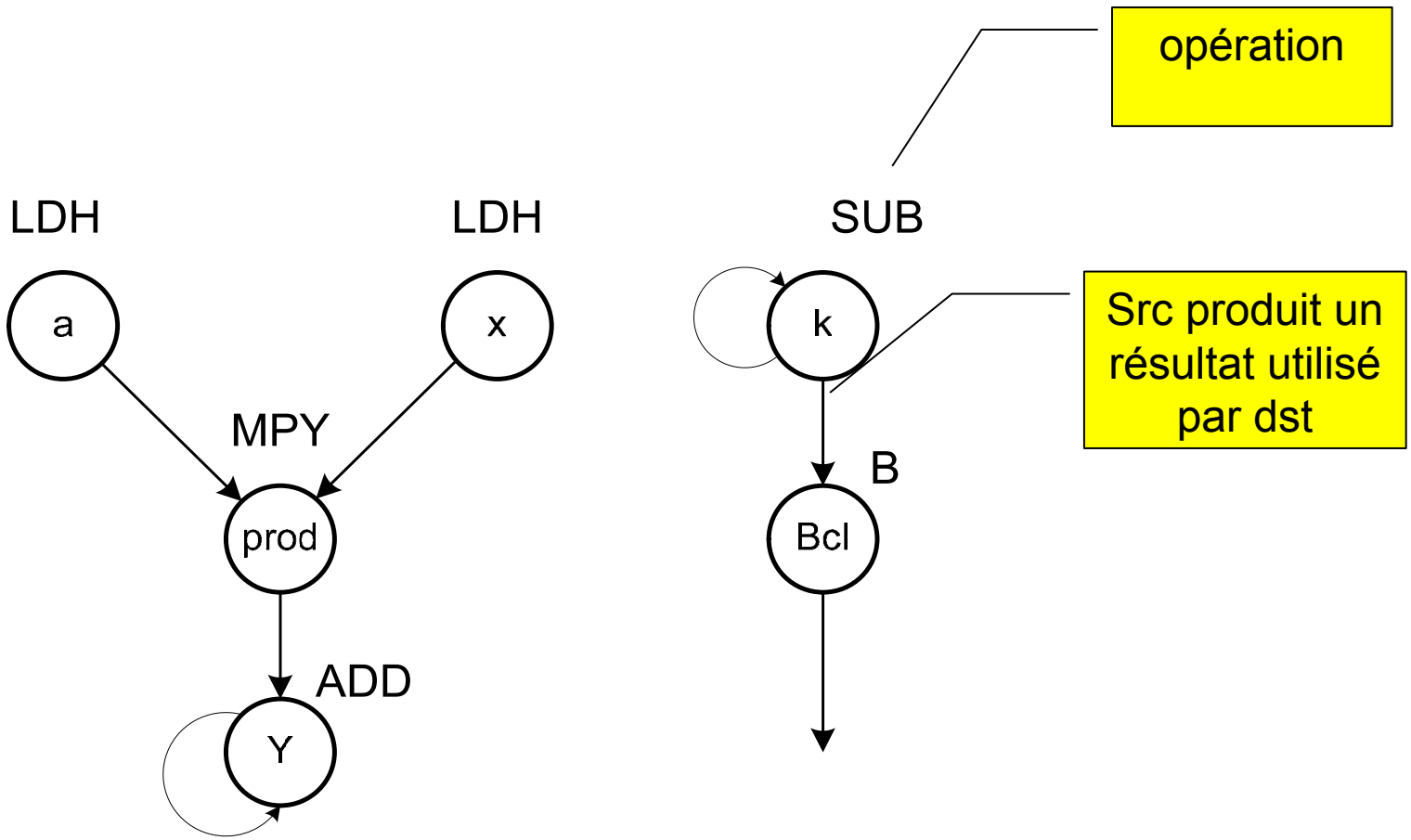
Ou comment écrire des programmes qui tirent parti de l'architecture pipeline du processeur?

Méthode:

1. Construire le graphe des dépendances
2. Partitionner
3. Ordonnancer

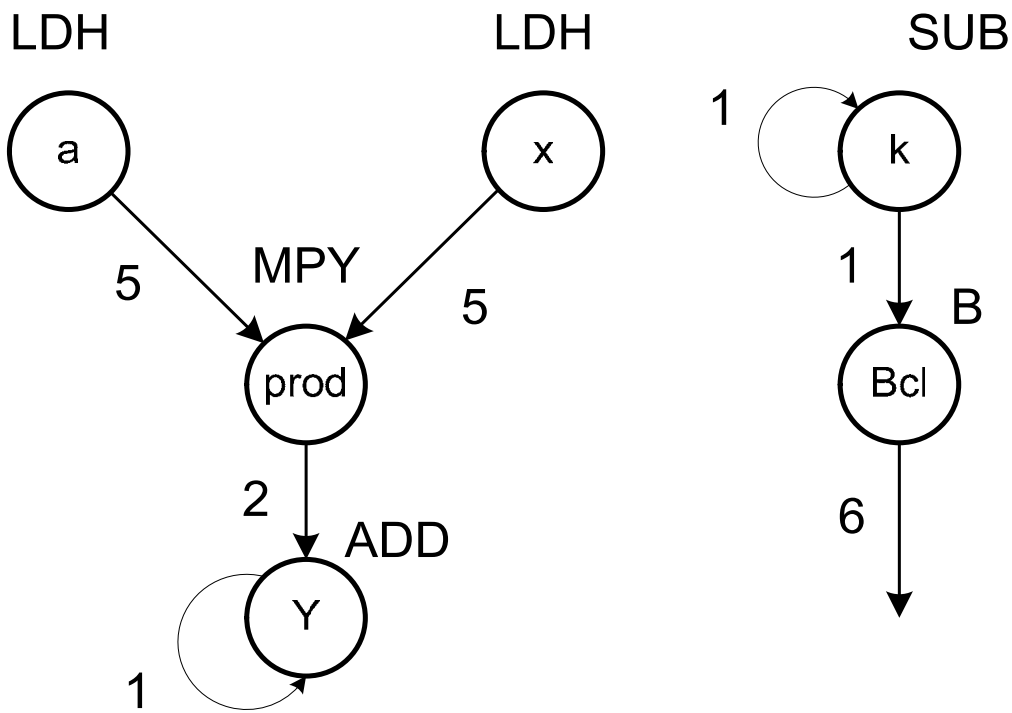
Graphe de dépendance

Exemple du produit scalaire

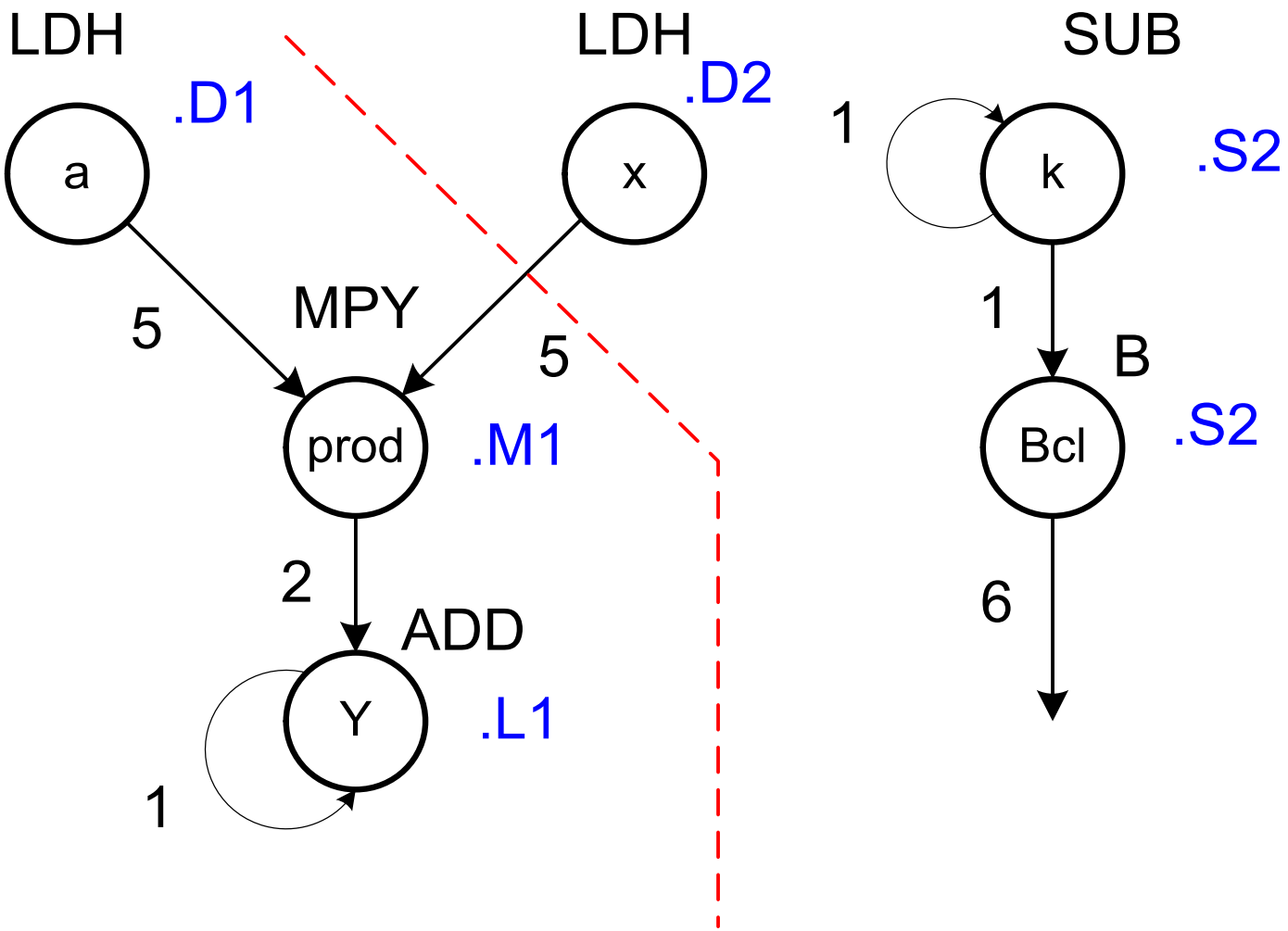


Graphe de dépendance (2)

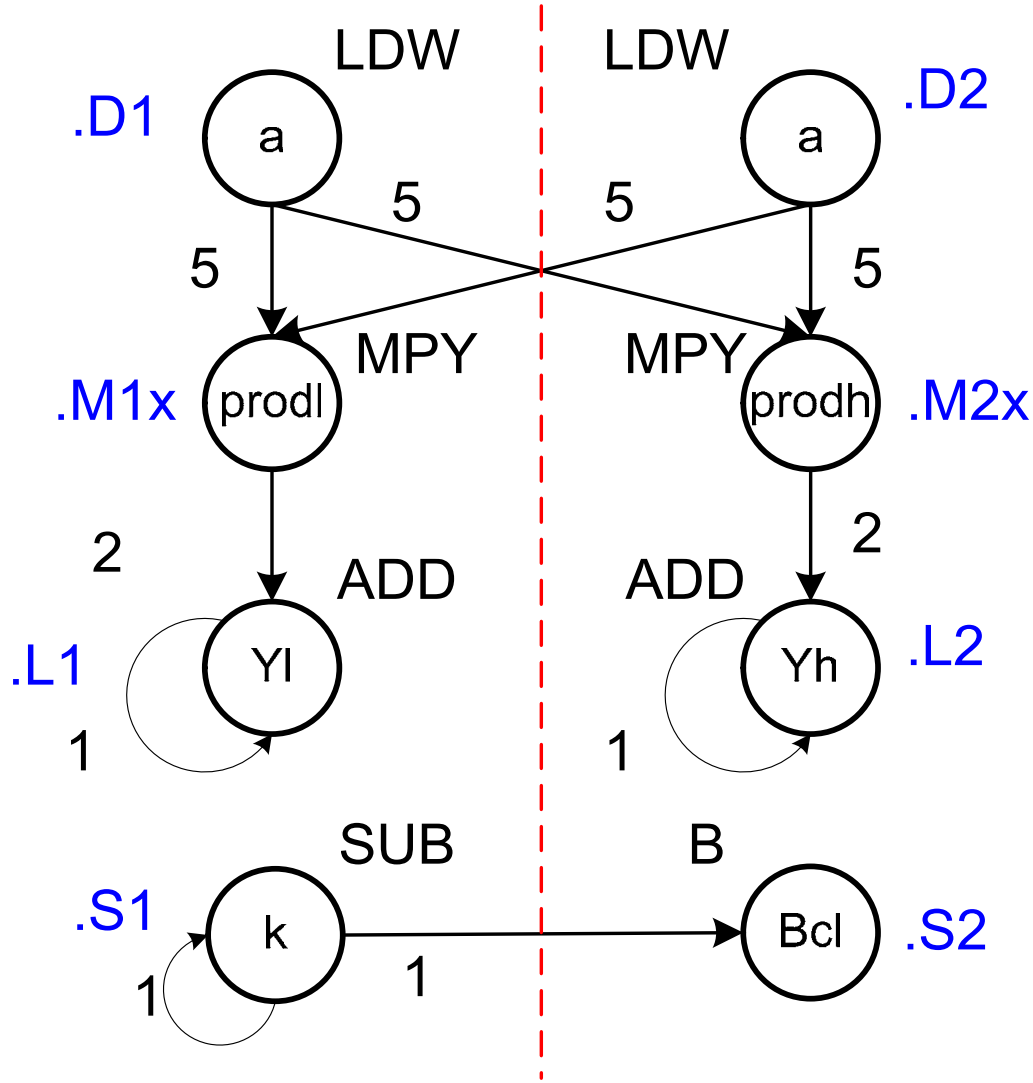
Ajouter les informations de durées



Partitionnement



Produit scalaire sur Words



Ordonnancement

Il y a trois modes de fonctionnement différents dans un pipeline :

1. Le prologue : contient les instructions nécessaires pour amorcer le pipeline.
2. Le régime permanent : correspondant à l'exécution du corps de boucle à la vitesse maximale.
3. L'épilogue : vidange du pipeline

Table d'ordonnancement

	1	2	3	4	5	6	7	8
.D1	LDW							
.D2	LDW							
.M1						MPY		
.M2						MPYH		
.L1								ADD
.L2								ADD
.S1		SUB						
.S2			B					

Table d'ordonnancement

	1	2	3	4	5	6	7	8
.D1	LDW	LDW	LDW	LDW	LDW	LDW	LDW	LDW
.D2	LDW	LDW	LDW	LDW	LDW	LDW	LDW	LDW
.M1						MPY	MPY	MPY
.M2						MPYH	MPYH	MPYH
.L1								ADD
.L2								ADD
.S1		SUB	SUB	SUB	SUB	SUB	SUB	SUB
.S2			B	B	B	B	B	B

Performance du pipeline

- Le prologue dure 7 cycles (longueur du plus long chemin dans le graphe de dépendance – 1)
- Le régime permanent dure 100 cycles
- L'épilogue nécessite un cycle
- Soit 108 cycles à comparer à la meilleures des optimisations précédentes (800 cycles)
- Conclusion : pour des calculs avec nombreuses itérations les performances sont considérablement accrues en programmant le pipeline, mais bien sûr ceci demande du travail!

Programme assembleur

```
;cycle 1
                MVK     .S1     100,A1          ;loop count
                ZERO    .L1     A4              ;init accum A4
   ||           ZERO    .L2     B2              ;init accum B2
   ||           LDW     .D1     *A8++,A2        ;32-bit data in A2
   ||           LDW     .D2     *B9++,B3        ;32-bit data in B3
;cycle 2
                LDW     .D1     *A8++,A2        ;32-bit data in A2
   ||           LDW     .D2     *B9++,B3        ;32-bit data in B3
   || [A1]      SUB     .S1     A1,1,A1         ;decrement count
;cycle 3
                LDW     .D1     *A8++,A2        ;32-bit data in A2
   ||           LDW     .D2     *B9++,B2        ;32-bit data in B3
   || [A1]      SUB     .S1     A1,1,A1         ;decrement count
   || [A1]      B       .S2     LOOP            ;branch to LOOP
```

Programme assembleur

```
;cycles 8-107 (loop cycle)
                LDW   .D1    *A8++,A2      ;32-bit data in A2
   ||           LDW   .D2    *B9++,B3      ;32-bit data in B3
   || [A1]      SUB   .S1    A1,1,A1       ;decrement count
   || [A1]      B     .S2    LOOP          ;branch to LOOP
   ||           MPY   .M1x   A2,B3,A6      ;lower 16-bit prod
   ||           MPYH  .M2x   B3,A2,B6      ;upper 16-bit prod
   ||           ADD   .L1    A6,A4,A4      ;accum in A4
   ||           ADD   .L2    B6,B2,B2      ;accum in B2
;branch occurs here
;cycle 108 (epilog)
         ADD   .L1x   A4,B2,A5      ;final accum of odd/even
```